

Лабораторная работа №2

Никитин Богдан, М3336

Вариант 11. Описание массивов в Kotlin

1 Разработка грамматики

Построим грамматику.

$$S \rightarrow \text{var Ident} : \text{Array} < \text{Ident} >$$

Нетерминал	Описание
S	Описание массива в Kotlin

2 Построение лексического анализатора

В нашей грамматике есть нетерминалы 'var', 'Ident', 'Array', '<', '>', ':', ';'. Построим лексический анализатор. Заведём класс TokenType для типа терминала и класс Token для хранения значения токена Ident. Не забудем также про конец строки.

```
package feo;

public enum TokenType {
    VAR("var"),
    ARRAY("Array"),
    LANGLEBRACKET("<"),
    RANGLEBRACKET(">"),
    IDENT("identifier"),
    END("end of file"),
    COLON(":"),
    SEMICOLON(";");

    private final String toString;

    TokenType(final String toString) {
        this.toString = toString;
    }

    public String toString() {
        return toString;
    }
}

package feo;

import java.util.Objects;

public class Token {
    private final TokenType tokenType;
    private final String value;

    public Token(TokenType tokenType, String value) {
        this.tokenType = tokenType;
        this.value = value;
    }
}
```

```

public Token(TokenType tokenType) {
    this(tokenType, null);
}

public String getValue() {
    return value;
}

public TokenType getTokenType() {
    return tokenType;
}

@Override
public boolean equals(final Object obj) {
    if (obj instanceof Token that) {
        return that.tokenType == tokenType && Objects.equals(that.value, value);
    }
    return false;
}

public final static Token VAR = new Token(TokenType.VAR);
public final static Token ARRAY = new Token(TokenType.ARRAY);
public final static Token LANGLEBRACKET = new Token(TokenType.LANGLEBRACKET);
public final static Token RANGLEBRACKET = new Token(TokenType.RANGLEBRACKET);
public final static Token END = new Token(TokenType.END);
public final static Token COLON = new Token(TokenType.COLON);
public final static Token SEMICOLON = new Token(TokenType.SEMICOLON);
}

```

Терминал	Токен
var	Token.VAR
Ident	new Token(TokenType.IDENT, "value")
Array	Token.ARRAY
<	Token.LANGLEBRACKET
>	Token.RANGLEBRACKET
:	Token.COLON
;	Token.SEMICOLON

Нжие представлен код лексического анализатора.

```

package feo;

import java.io.IOException;
import java.io.InputStream;
import java.text.ParseException;

public class LexicalAnalyzer {
    private final InputStream is;
    private int curChar;
    private int curPos;
    private Token curToken;

    public LexicalAnalyzer(final InputStream is) throws ParseException {

```

```

        this.is = is;
        curPos = 0;
        nextChar();
    }

    private boolean isBlank(final int c) {
        return c == ' ' || c == '\r' || c == '\n' || c == '\t';
    }

    private void nextChar() throws ParseException {
        curPos++;
        try {
            curChar = is.read();
        } catch (IOException e) {
            throw new ParseException(e.getMessage(), curPos);
        }
    }

    public void nextToken() throws ParseException {
        while (isBlank(curChar)) {
            nextChar();
        }
        switch (curChar) {
            case '<':
                nextChar();
                curToken = Token.LANGLEBRACKET;
                break;
            case '>':
                nextChar();
                curToken = Token.RANGLEBRACKET;
                break;
            case ';':
                nextChar();
                curToken = Token.SEMICOLON;
                break;
            case ':':
                nextChar();
                curToken = Token.COLON;
                break;
            case '-1':
                curToken = Token.END;
                break;
            default:
                final String ident = parseIdent();
                if (ident.equals(TokenType.VAR.toString())) {
                    curToken = Token.VAR;
                } else if (ident.equals(TokenType.ARRAY.toString())) {
                    curToken = Token.ARRAY;
                } else {
                    curToken = new Token(TokenType.IDENT, ident);
                }
        }
    }

    private void illegalCharacter() throws ParseException {
        throw new ParseException("Illegal character " + (char) curChar, curPos);
    }

```

```

    }

    private String parseIdent() throws ParseException {
        if (!Character.isLetter(curChar) && curChar != '_') {
            illegalCharacter();
        }
        final StringBuilder stringBuilder = new StringBuilder();
        do {
            stringBuilder.appendCodePoint(curChar);
            nextChar();
        } while (Character.isLetterOrDigit(curChar) || curChar == '_');
        return stringBuilder.toString();
    }

    public Token curToken() {
        return curToken;
    }

    public int curPos() {
        return curPos;
    }
}

```

3 Построение лексического анализатора

Построим множества FIRST и FOLLOW для нетерминалов нашей грамматики.

Нетерминал	FIRST	FOLLOW
<i>S</i>	var	\$

Заведем структуру данных для хранения результата.

```
package feo;
```

```
public record ArrayDeclaration(String variableName, String elementType) {}
```

Построим синтаксический анализатор.

```
package feo;
```

```
import java.io.InputStream;
import java.text.ParseException;
```

```
public class Parser {
    private LexicalAnalyzer lex;

    public Parser() {
    }

```

```

    public ArrayDeclaration parse(final InputStream is) throws ParseException {
        lex = new LexicalAnalyzer(is);
        expect(TokenType.VAR);
        expect(TokenType.IDENT);
        final String variableName = lex.curToken().getValue();
        expect(TokenType.COLON);
        expect(TokenType.ARRAY);
        expect(TokenType.LANGLEBRACKET);
        expect(TokenType.IDENT);
    }
}

```

```

        final String elementType = lex.curToken().getValue();
        expect(TokenType.RANGLEBRACKET);
        expect(TokenType.SEMICOLON);
        expect(TokenType.END);
        return new ArrayDeclaration(variableName, elementType);
    }

    private void expect(final TokenType tokenType) throws ParseException {
        lex.nextToken();
        if (lex.curToken().getTokenType() != tokenType) {
            throw new ParseException(tokenType.toString() + " expected at position",
lex.curPos());
        }
    }
}

```

4 Визуализация дерева разбора

Выведём в формате DOT значения имени массива и имени типа элементов массива.

```

package feo;

import java.text.ParseException;

public class Main {
    public static void main(String[] args) {
        try {
            final ArrayDeclaration array = new Parser().parse(System.in);
            System.out.printf("digraph { ArrayDeclaration -> \"%s\" ArrayDeclaration -
> \"%s\" }%n",
                array.variableName(),
                array.elementType());
        } catch (ParseException e) {
            System.err.println("During parsing an error occurred: " + e);
        }
    }
}

```

5 Подготовка набора тестов

Описание теста приведено в аннотации DisplayName.

Тексты к лексическому анализатору.

```

package feo;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.io.ByteArrayInputStream;
import java.nio.charset.StandardCharsets;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;

class LexicalAnalyzerTest {
    final Token[] SIMPLE_TOKENS = {
        Token.VAR,

```

```

        Token.ARRAY,
        Token.LANGLEBRACKET,
        Token.RANGLEBRACKET,
        Token.COLON,
        Token.SEMICOLON
    };

    LexicalAnalyzer run(final String input) throws ParseException {
        return new LexicalAnalyzer(new
        ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8)));
    }

    List<Token> tokens(final String input) throws ParseException {
        final LexicalAnalyzer lex = run(input);
        final List<Token> tokenList = new ArrayList<>();
        while (true) {
            lex.nextToken();
            if (lex.curToken() == Token.END) {
                break;
            }
            tokenList.add(lex.curToken());
        }
        return tokenList;
    }

    void checkSequence(final String input, final Token... tokens) throws ParseException
    {
        Assertions.assertArrayEquals(tokens(input).toArray(), tokens);
    }

    Token ident(final String value) {
        return new Token(TokenType.IDENT, value);
    }

    void checkIdent(final String ident) throws ParseException {
        checkSequence(ident, ident(ident));
    }

    void checkException(final String input) {
        Assertions.assertThrows(ParseException.class, () -> checkSequence(input));
    }

    @Test
    @DisplayName("Empty input")
    void testEmpty() throws ParseException {
        checkSequence("");
    }

    @Test
    @DisplayName("Input of one single token without value")
    void testSimpleTokens() throws ParseException {
        for (Token token : SIMPLE_TOKENS) {
            checkSequence(token.getTokenType().toString(), token);
        }
    }
}

```

```

@Test
@DisplayName("Check identifier in many variants")
void testIdent() throws ParseException {
    checkIdent("foobar");
    checkIdent("FooBar");
    checkIdent("FooBar");
    checkIdent("_foobar");
    checkIdent("_____");
    checkIdent("foobar_");
    checkIdent("foo_bar");
    checkIdent("_foo_bar_");
    checkIdent("foo42");
    checkIdent("foo42bar");
}

@Test
@DisplayName("Case of var and Array")
void testCase() throws ParseException {
    checkIdent("Var");
    checkIdent("vAr");
    checkIdent("vaR");
    checkIdent("VAR");
    checkIdent("ArraY");
    checkIdent("array");
    checkIdent("ARRAY");
}

@Test
@DisplayName("Prefixes of var and Array to be an identifier")
void testPrefix() throws ParseException {
    checkIdent("v");
    checkIdent("va");
    checkIdent("varfoo");
    checkIdent("A");
    checkIdent("Arra");
    checkIdent("Arraybar");
}

@Test
@DisplayName("Check array declaration")
void testValid() throws ParseException {
    checkSequence("var foobar42: Array<Int>;",
        Token.VAR, ident("foobar42"), Token.COLON, Token.ARRAY,
        Token.LANGLEBRACKET, ident("Int"), Token.RANGLEBRACKET, Token.SEMICOLON);
}

@Test
@DisplayName("Check random valid tokens")
void testRandom() throws ParseException {
    checkSequence("Array>foobar<;:var",
        Token.ARRAY, Token.RANGLEBRACKET, ident("foobar"), Token.LANGLEBRACKET,
        Token.SEMICOLON, Token.SEMICOLON, Token.COLON, Token.VAR
    );
}

@Test

```

```

@DisplayName("Check blank characters")
void testBlank() throws ParseException {
    checkSequence(" ");
    checkSequence("\r");
    checkSequence("\n");
    checkSequence("\t");
    checkSequence(" \r \n\t\n\n \r\r\t \n\r");
    checkSequence(" \r \n\n\t \r var \r\n \t foobar42 \n \r \r\r : \n
\rArray\r<\tInt > \r; \n",
        Token.VAR, ident("foobar42"), Token.COLON, Token.ARRAY,
        Token.LANGLEBRACKET, ident("Int"), Token.RANGLEBRACKET, Token.SEMICOLON);
}

@Test
@DisplayName("Identifier starting with number")
void testDigitFirstIdentifier() {
    checkException("42foobar");
    checkException("var 42foobar: Array<Int>");
    checkException("var foobar: Array<42Int>");
}

@Test
@DisplayName("Invalid characters")
void testInvalidCharacters() {
    checkException("var foobar: Array<Int+>");
    checkException("var$ foobar: Array<Int>");
    checkException("var foobar: Array<Int>;%");
}
}

```

Тексты к синтаксическому анализатору.

```

package feo;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.io.ByteArrayInputStream;
import java.nio.charset.StandardCharsets;
import java.text.ParseException;

class ParserTest {
    ArrayDeclaration parse(final String input) throws ParseException {
        return new Parser().parse(new
        ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8)));
    }

    void checkDeclaration(final String input, final String variableName, final String
elementType) throws ParseException {
        Assertions.assertEquals(parse(input), new ArrayDeclaration(variableName,
elementType));
    }

    void checkException(final String input) {
        Assertions.assertThrows(ParseException.class, () -> parse(input));
    }
}

```



```

@Test
@DisplayName("Simple test for valid input")
void testValid() throws ParseException {
    checkDeclaration("var foo: Array<Int>;", "foo", "Int");
    checkDeclaration("var foo:Array<Z>;", "foo", "Z");
    checkDeclaration("var bar : Array<kgeorgiy>;", "bar", "kgeorgiy");
}

@Test
@DisplayName("Test blank characters")
void testBlank() throws ParseException {
    checkDeclaration("\n\tvar \n\r\tfoo\n\t : Array < Int > ; ", "foo",
"Int");
}

@Test
@DisplayName("Test valid case with trailing characters")
void testTrailing() {
    checkException("var foo: Array<Int>;");
    checkException("var foo: Array<Int>;var");
    checkException("var foo: Array<Int>;bar");
}

@Test
@DisplayName("Test missing tokens")
void testMissing() {
    checkException("foo: Array<Int>;");
    checkException("var: Array<Int>;");
    checkException("var foo Array<Int>;");
    checkException("var foo: <Int>;");
    checkException("var foo: Array Int>;");
    checkException("var foo: Array<>;");
    checkException("var foo: Array<Int>;");
    checkException("var foo: Array<Int>");
}

@Test
@DisplayName("Array as element type")
void testArrayOfArrays() {
    checkException("var array: Array<Array>;");
}

@Test
@DisplayName("Test repeating tokens")
void testRepeats() {
    checkException("var var foo: Array<Int>;");
    checkException("var foo bar: Array<Int>;");
    checkException("var foo: : Array<Int>;");
    checkException("var foo: Array Array<Int>;");
    checkException("var foo: Array<<Int>;");
    checkException("var foo: Array<Int>>;");
    checkException("var foo: Array<Int>;");
}

@Test
@DisplayName("Wrong brackets")

```

```
void testBrackets() {  
    checkException("var foo: Array>Int>");  
    checkException("var foo: Array<Int<");  
    checkException("var foo: Array>Int<");  
}  
}
```